Outline

- Parallel programming models
- Dependencies
- Structured programming patterns overview
  - Serial / parallel control flow patterns
  - Serial / parallel data management patterns
Parallel Models 101

- Sequential models
  - von Neumann (RAM) model

- Parallel model
  - A parallel computer is simply a collection of *processors interconnected* in some manner to *coordinate* activities and *exchange data*
  - Models that can be used as general frameworks for describing and analyzing parallel algorithms
    - *Simplicity*: description, analysis, architecture independence
    - *Implementability*: able to be realized, reflect performance

- Three common parallel models
  - Directed acyclic graphs, shared-memory, network
Directed Acyclic Graphs (DAG)

- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as tasks to be performed
- Arcs are paths for flow of data (in and out of tasks)
- DAG represents the algorithm operations and implies precedent constraints on their order of execution

```c
for (i=1; i<100; i++)
a[i] = a[i-1] + 100;
```

- Where is the parallelism?
  - Think about when a node can execute

Can this DAG be executed in parallel?
Shared Memory Model

- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors in unbounded
  - Processors communicate via the memory
  - Every processor accesses in 1 cycle
  - Synchronous execution
    - all processors execute same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - some subset of processors can stay idle (implicit synchronization)
  - Asynchronous execution
    - processors run independently
    - synchronization using memory
Network Model

- $G = (N,E)$
  - $N$ are processing nodes
  - $E$ are bidirectional communication links
  - Edges define a network “topology”
- Each processor has its own memory
- No shared memory is available
- Network operation
  - Synchronous or asynchronous
- Requires communication primitives
  - Send $(X, i)$
  - Receive $(Y, j)$
- Captures message passing model for algorithm design
Parallelism

- Ability to execute different parts of a computation concurrently on different machines

- Why do you want parallelism?
  - Shorter running time (time to solution)
  - Handling more work (bigger problems, throughput)

- What is being parallelized?
  - Task: instruction, statement, procedure, …
  - Data: data flow, size, replication
  - Parallelism granularity (both in task and data)
    - coarse-grain versus fine-grained

- Thinking about parallelism (concurrency, efficiency)

- Evaluation
Why is parallel programming important?

- Parallel programming has matured
  - Standard programming models
  - Common machine architectures
  - Programmer can focus on computation and use suitable programming model for implementation

- Increase portability between models and architectures

- Reasonable hope of portability across platforms

- Problem
  - Performance optimization is still platform-dependent
  - Performance portability is a problem
  - Parallel programming methods are still evolving
Parallel Algorithm

- Recipe to solve a problem “in parallel” on multiple processing elements
- Standard steps for constructing a parallel algorithm
  - Identify work that can be performed concurrently
  - Partition the concurrent work on separate processors
  - Properly manage input, output, and intermediate data
  - Coordination to satisfy dependencies
    - data consistency
  - Coordination to manage parallelism
- Which are hard to do?
Parallelism Views

- Where can we find parallelism?
- Program (task) view
  - Statement level
    - Between program statements
    - Which statements can be executed at the same time?
  - Block level / Loop level / Routine level / Process level
    - Larger-grained program statements
- Data view
  - How is data operated on?
  - Where does data reside?
- Resource view
Parallel execution, from any point of view, will be constrained by the sequence of operations needed to be performed for a correct result.

Parallel execution must address control, data, and system dependences.

A dependency arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed.

We extend this notion of dependency to resources since some operations may depend on certain resources:
- Due to where data is located
- Due to what processing resources might be available
Executing Two Statements in Parallel

- Want to execute two statements in parallel
  - Execute at the same time on different processors

- On one processor:
  Statement 1;
  Statement 2;

- On two processors:
  Processor 1:  Processor 2:
  Statement 1;  Statement 2;

- Fundamental (concurrent) execution assumption
  - Processors execute independent of each other
  - No assumptions made about speed of processor execution
    - why do we care?
Sequential Consistency in Parallel Execution

- **Case 1:**
  - Processor 1: Statement 1;
  - Processor 2: Statement 2;

- **Case 2:**
  - Processor 1: Statement 2;
  - Processor 2: Statement 1;

- **Sequential consistency**
  - Statements execution does not interfere with each other
  - Computation results are the same (independent of order)
Independent versus Dependent

- In other words, the execution of
  
  Statement1;
  
  Statement2;

  must be equivalent to (i.e., producing the same outcome)
  
  Statement2;
  
  Statement1;

- Sequential consistency means that the statement order of execution must not matter!

- If true, the statements are *independent* of each other

- Two statements are *dependent* when the order of their execution affects the computation outcome
Examples

- **Example 1**
  
  $S_1$: $a = 1$;
  $S_2$: $b = 1$;

- **Example 2**
  
  $S_1$: $a = 1$;
  $S_2$: $b = a$;

- **Example 3**
  
  $S_1$: $a = f(x)$;
  $S_2$: $a = b$;

- **Example 4**
  
  $S_1$: $a = b$;
  $S_2$: $b = 1$;

Questions:

- Independent or dependent?
  
  $(a = 5, b = 6)$ $S_1$ then $S_2$ $(a = 1, b = 1)$
  
  $(a = 5, b = 6)$ $S_2$ then $S_1$ $(a = 1, b = 1)$

- Independent or dependent?
  
  $(a = 5, b = 6)$ $S_1$ then $S_2$ $(a = 1, b = 5)$
  
  $(a = 5, b = 6)$ $S_2$ then $S_1$ $(a = 1, b = 1)$

- Independent or dependent?
  
  $(a = 5, b = 6)$ $S_1$ then $S_2$ $(a = f(x))$
  
  $(a = 5, b = 6)$ $S_2$ then $S_1$ $(a = f(x))$

- Independent or dependent?
  
  $(a = 5, b = 6)$ $S_1$ then $S_2$ $(a = 6, b = 1)$
  
  $(a = 5, b = 6)$ $S_2$ then $S_1$ $(a = 1, b = 1)$
Examples

- **Example 1**
  
  \[ \begin{align*}
  S_1 & : a=1; \\
  S_2 & : b=1;
  \end{align*} \]

- **Example 2**
  
  \[ \begin{align*}
  S_1 & : a=1; \\
  S_2 & : b=a;
  \end{align*} \]

- **Example 3**
  
  \[ \begin{align*}
  S_1 & : a=f(x); \\
  S_2 & : a=b;
  \end{align*} \]

- **Example 4**
  
  \[ \begin{align*}
  S_1 & : a=b; \\
  S_2 & : b=1;
  \end{align*} \]

  - Statements are independent
  - Dependent (*true (flow) dependence*)
    - Second is dependent on first
    - Can you remove dependency?
  - Dependent (*output dependence*)
    - Second is dependent on first
    - Can you remove dependency? How?
  - Dependent (*anti-dependence*)
    - First is dependent on second
    - Can you remove dependency? How?
True Dependence and Anti-Dependence

- Given statements $S_1$ and $S_2$,
  
  $S_1$;
  
  $S_2$;

- $S_2$ has a \textit{true (flow) dependence} on $S_1$
  
  if and only if
  
  $S_2$ reads a value written by $S_1$

- $S_2$ has a \textit{anti-dependence} on $S_1$
  
  if and only if
  
  $S_2$ writes a value read by $S_1$
Output Dependence

- Given statements $S_1$ and $S_2$,
  
  $S_1;$
  
  $S_2;$

- $S_2$ has an *output dependence* on $S_1$ if and only if
  
  $S_2$ writes a variable written by $S_1$

- Anti- and output dependences are “name” dependencies
  
  - Are they “true” dependences?

- How can you get rid of output dependences?
  
  - Are there cases where you can not?
Statement Dependency Graphs

- Can use graphs to show dependence relationships
- Example

  S_1: a=1;
  S_2: b=a;
  S_3: a=b+1;
  S_4: c=a;

- S_2 \delta S_3 : S_3 is flow-dependent on S_2
- S_1 \delta^0 S_3 : S_3 is output-dependent on S_1
- S_2 \delta^{-1} S_3 : S_3 is anti-dependent on S_2
When can two statements execute in parallel?

- Statements $S_1$ and $S_2$ can execute in parallel if and only if there are *no dependences* between $S_1$ and $S_2$
  - True dependences
  - Anti-dependences
  - Output dependences

- Some dependences can be removed by modifying the program
  - Rearranging statements
  - Eliminating statements

- Dependence analysis came from compilers looking for opportunities for improving ILP
How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node.
- The IN and OUT sets of a statement S are defined as:
  - \( \text{IN}(S) \) : set of memory locations (variables) that may be used in S
  - \( \text{OUT}(S) \) : set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified.
- As such, the sets can be conservatively large.
IN / OUT Sets and Computing Dependence

- Assuming that there is a path from $S_1$ to $S_2$, the following shows how to intersect the IN and OUT sets to test for data dependence

\[
\begin{align*}
\text{out}(S_1) \cap \text{in}(S_2) &\neq \emptyset & S_1 &\delta S_2 & \text{flow dependence} \\
\text{in}(S_1) \cap \text{out}(S_2) &\neq \emptyset & S_1 &\delta^{-1} S_2 & \text{anti-dependence} \\
\text{out}(S_1) \cap \text{out}(S_2) &\neq \emptyset & S_1 &\delta^0 S_2 & \text{output dependence}
\end{align*}
\]
Loop-Level Parallelism

- Significant parallelism can be identified within loops

```cpp
for (i=0; i<100; i++)
    S_1: a[i] = i;
```

- Dependencies? What about `i`, the loop index?

- **DOALL** loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements be executed in parallel at the same time
    - is this really true?

- Imagine that the loop is *unrolled*
Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations

```plaintext
# Iteration Space

for (i=0; i<100; i++)

\[ S_1: a[i] = i; \]

for (i=0; i<100; i++) {
    \[ S_1: a[i] = i; \]
    \[ S_2: b[i] = 2*i; \]
}
```
Multi-Loop Parallelism

- Significant parallelism can be identified between loops

```c
for (i=0; i<100; i++) a[i] = i;
```

```c
for (i=0; i<100; i++) b[i] = i;
```

- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?
Multi-Loop Parallelism

- Significant parallelism can be identified between loops

```plaintext
for (i=0; i<100; i++) a[i] = i;
c = 5
for (i=0; i<100; i++) b[i] = c;
```

- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?
Loops with Dependencies

Case 1:
for (i=1; i<100; i++)
a[i] = a[i-1] + 100;

 Dependencies?
○ What type?

Is the Case 1 loop parallelizable?

Is the Case 2 loop parallelizable?

Case 2:
for (i=5; i<100; i++)
a[i-5] = a[i] + 100;
Another Loop Example

```c
for (i=1; i<100; i++)
a[i] = f(a[i-1]);
```

- Dependencies?
  - What type?
- Loop iterations are not parallelizable
  - Why not?
Loop Dependencies

- A *loop-carried* dependence is a dependence that is present only if the statements are part of the execution of a loop (i.e., between two statements instances in two different iterations of a loop)
- Otherwise, it is *loop-independent*, including between two statements instances in the same loop iteration
- Loop-carried dependences can prevent loop iteration parallelization
- The dependence is *lexically forward* if the source comes before the target or *lexically backward* otherwise
  - Use *loop unrolling* to see
Loop Dependence Example

```
for (i=0; i<100; i++)
    a[i+10] = f(a[i]);
```

- Dependencies?
  - Between a[10], a[20], …
  - Between a[11], a[21], …

- Some parallel execution is possible
  - How much?
for (i=1; i<100; i++) {
    S1: a[i] = …;
    S2: … = a[i-1];
}

- Dependencies?
  - Between a[i] (S1) and a[i-1] (S2)

- Is parallelism possible?
  - Statements can be executed in “pipeline” manner
  - What else?
Another Loop Dependence Example

```c
for (i=0; i<100; i++)
    for (j=1; j<100; j++)
        a[i][j] = f(a[i][j-1]);
```

- Dependencies?
  - Loop-independent dependence on $i$
  - Loop-carried dependence on $j$

- Which loop can be parallelized?
  - Outer loop parallelizable
  - Inner loop cannot be parallelized
Still Another Loop Dependence Example

for (j=1; j<100; j++)
    for (i=0; i<100; i++)
        a[i][j] = f(a[i][j-1]);

- Dependencies?
  - Loop-independent dependence on \(i\)
  - Loop-carried dependence on \(j\)

- Which loop can be parallelized?
  - Inner loop parallelizable
  - Outer loop cannot be parallelized
  - Less desirable (why?)
Key Ideas for Dependency Analysis

- To identify parallelism opportunities:
  - Statement order must not matter
  - Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious
Dependencies and Synchronization

- How is parallelism achieved when have dependencies?
  - Think about concurrency
  - Some parts of the execution are independent
  - Some parts of the execution are dependent

- Must control ordering of events on different processors (cores)
  - Dependencies pose constraints on parallel event ordering
  - Partial ordering of execution action

- Use synchronization mechanisms
  - Need for concurrent execution too
  - Maintains partial order
Parallel Patterns

- A parallel pattern is a recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a “vocabulary” for algorithm design.
- It can be useful to compare parallel patterns with serial patterns.
- Patterns are universal
  - They can be used in any parallel programming system.
Parallel Patterns Overview

- Nesting Pattern
- Serial / Parallel Control Patterns
- Serial / Parallel Data Management Patterns
- Other Patterns
- Programming Model Support for Patterns
Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns.
- This pattern appears in both serial and parallel algorithms.
- *Pattern diagrams* are used to visually show the pattern idea where each *task block* is a location of general code in an algorithm.
- Each task block can in turn be another pattern in the *nesting pattern*.
**Nesting Pattern**: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.
Serial Control Patterns

- Structured serial programming is based on **serial control patterns** that order statement execution (*imperative* programming):
  - Sequence
  - Selection
  - Iteration
  - Recursion

- The **nesting** pattern can also be used to hierarchically compose these four patterns

- Though you should be familiar with these, it’s extra important to understand these patterns when parallelizing serial algorithms based on these them
Serial Control Patterns: Sequence

- Ordered list of tasks that are executed in a specific order constitute a **sequence** control pattern.

Assumption:
- Program text ordering will be followed (obvious, but this will be important when parallelized)

```
1  T = f(A);
2  S = g(T);
3  B = h(S);
```

```
1  T = f(A);
2  S = g(A);
3  B = h(S,T);
```
Serial Control Patterns: Selection

- A selection control pattern evaluates a condition $c$ is first, and then executes task $a$ or $b$ depending on the true or false result of $c$

- Assumptions:
  - $a$ and $b$ are never executed before $c$, and only $a$ or $b$ is executed - never both

```java
1 if (c) {
2    a;
3  } else {
4    b;
5  }
```
Serial Control Patterns: Iteration

- With the **iteration** control pattern, a condition \( c \) is evaluated. If true, \( a \) is evaluated, and then \( c \) is evaluated again. This repeats until \( c \) is false.

- Complication when parallelizing:
  - Potential for dependencies to exist between previous iterations

```c
1  for (i = 0; i < n; 
2       a; 
3  }

1  while (c) {
2     a; 
3  }
```
Serial Control Patterns: Recursion

- The **recursion** control pattern is a dynamic form of nesting allowing functions to call themselves.
- Tail recursion is a special recursion that can be converted into iteration.
  - Particularly important for functional languages.
Parallel Control Patterns

- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns
- Parallel control patterns:
  - Fork-Join
  - Map
  - Stencil
  - Reduction
  - Scan
  - Recurrence
Parallel Control Patterns: Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later.
- **Cilk Plus** implements this with `spawn` and `sync`:
  - Calling tree is a parallel call tree and functions are spawned instead of being called.
  - Functions that spawn another function call will continue to execute.
  - Caller `syncs` with the spawned function to join the two.
- A “join” is different than a “barrier”:
  - Sync – only one thread continues.
  - Barrier – all threads continue.
Parallel Control Patterns: Map

- **Map**: performs a function over every element of a collection
  - Replicates a serial iteration pattern
  - Each iteration is independent of the others
  - Number of iterations is known in advance
  - Computation only depends on iteration count and data from input collection

- The replicated function is referred to as an “elemental function”
Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of “neighbors”, stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time
- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture…
Parallel Control Patterns: Reduction

- **Reduction**: Combines every element in a collection using an associative “combiner function”

- Because of the associativity of the combiner function, different orderings of the reduction are possible

- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR
Parallel Control Patterns: Reduction

Serial Reduction

Parallel Reduction
Parallel Control Patterns: Scan

- **Scan**: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop
- A parallel scan will require more operations than a serial version
Parallel Control Patterns: Scan

Serial Scan

Parallel Scan
Parallel Control Patterns: Recurrence

- **Recurrence**: More complex version of map, where the loop iterations can depend on one another.
- Similar to map, but elements can use outputs of adjacent elements as inputs.
- For a recurrence to be computable, there **must** be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs.
Serial Data Management Patterns

- Serial programs can manage data in many ways
- Data management deals with how data is allocated, shared, read, written, and copied
- Serial Data Management Patterns:
  - Random read and write
  - Stack allocation
  - Heap allocation
  - Objects
Serial Data Management Patterns: Read/Write

- Random memory addressing
- Memory locations indexed with addresses
- Pointers are typically used to refer to memory addresses
- Aliasing (uncertainty of two pointers referring to the same object) can cause problems when serial code is parallelized
Serial Data Management Patterns: Stack Allocation

- Stack allocation is useful for dynamically allocating data in LIFO manner
- Efficient
  - Arbitrary amount of data can be allocated in constant time
- Stack allocation also preserves locality
- When parallelized, typically each thread will get its own stack so thread locality is preserved
Serial Data Management Patterns: Heap Allocation

- Heap allocation is useful when data cannot be allocated in a LIFO fashion.
- But, heap allocation is slower and more complex than stack allocation.
- A parallelized heap allocator should be used when dynamically allocating memory in parallel.
  - This type of allocator will keep separate pools for each parallel worker.
Serial Data Management Patterns: Objects

- Objects are language constructs to associate data with code to manipulate and manage that data.
- Objects can have member functions, and they also are considered members of a class of objects.
- Parallel programming models will generalize objects in various ways.
Parallel Data Management Patterns

- To avoid things like race conditions, it is critically important to know when data is, and isn’t, potentially shared by multiple parallel workers.

- Some parallel data management patterns help us with data locality.

- Parallel data management patterns:
  - Pack
  - Pipeline
  - Geometric decomposition
  - Gather and scatter
Parallel Data Management Patterns: Pack

- **Pack** is used to eliminate unused space in a collection.
- Elements marked *false* are discarded, the remaining elements are placed in a contiguous sequence in the same order.
- Useful when used with **map**.
- **Unpack** is the inverse and is used to place elements back in their original locations.
Parallel Data Management Patterns: Pipeline

- **Pipeline** connects tasks in a producer-consumer manner.
- A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible.
- Pipelines are most useful when used with other patterns as they can multiply available parallelism.
Parallel Data Management Patterns: Decomposition

- (Geometric) Decomposition arranges data into subcollections
- Overlapping and non-overlapping decompositions are possible
- This pattern doesn’t necessarily move data, it just gives us another view of it
Parallel Data Management Patterns: Gather

- **Gather** reads a collection of data given a collection of indices.
- Think of a combination of map and random serial reads.
- The output collection shares the same type as the input collection, but it share the same shape as the indices collection.

![Diagram of Gather pattern](image)
Parallel Data Management Patterns: Scatter

- **Scatter** is the inverse of gather
- A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index
- Race conditions can occur when we have two writes to the same location!

![Diagram](image-url)
Other Parallel Patterns

- **Superscalar Sequences**: write a sequence of tasks, ordered only by dependencies
- **Futures**: similar to fork-join, but tasks do not need to be nested hierarchically
- **Speculative Selection**: general version of serial selection where the condition and both outcomes can all run in parallel
- **Workpile**: general map pattern where each instance of elemental function can generate more instances, adding to the “pile” of work
Other Parallel Patterns

- **Search**: finds some data in a collection that meets some criteria
- **Segmentation**: operations on subdivided, non-overlapping, non-uniformly sized partitions of 1D collections
- **Expand**: a combination of pack and map
- **Category Reduction**: Given a collection of elements each with a label, find all elements with same label and reduce them
Table 3.1 Summary of programming model support for the serial patterns discussed in this book. Note that some of the parallel programming models we consider do not, in fact, support all the common serial programming patterns. In particular, note that recursion and memory allocation are limited on some model.

<table>
<thead>
<tr>
<th>Serial Pattern</th>
<th>TBB</th>
<th>Cilk Plus</th>
<th>OpenMP</th>
<th>ArBB</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Serial) Nesting</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Sequence</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Selection</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Iteration</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Recursion</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>?</td>
</tr>
<tr>
<td>Random Read</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Random Write</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Stack Allocation</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Heap Allocation</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Closures</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>Objects</td>
<td>F</td>
<td>F</td>
<td>F(w/C++)</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
### Table 3.2: Summary of programming model support for the patterns discussed in this book.

<table>
<thead>
<tr>
<th>Parallel Pattern</th>
<th>TBB</th>
<th>Cilk Plus</th>
<th>OpenMP</th>
<th>ArBB</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map</td>
<td>F; 4.2.3;</td>
<td>F; 4.2.6;</td>
<td>4.3.6;</td>
<td>4.3.7;</td>
<td>4.3.8</td>
</tr>
<tr>
<td>Stencil</td>
<td>I; 10</td>
<td>I; 10</td>
<td>I; F.10</td>
<td>I;</td>
<td></td>
</tr>
<tr>
<td>Workpile</td>
<td>F; 5.3.4</td>
<td>F; 5.3.5</td>
<td>F; 5.3.6;</td>
<td>F; 5.3.7;</td>
<td></td>
</tr>
<tr>
<td>Reduction</td>
<td>4.3.3;</td>
<td>4.3.4;</td>
<td>4.3.5;</td>
<td>4.3.6;</td>
<td></td>
</tr>
<tr>
<td>Scan</td>
<td>11</td>
<td>11</td>
<td>I; 5.6.4;</td>
<td>F; 5.6.6;</td>
<td></td>
</tr>
<tr>
<td>Fork-join</td>
<td>14</td>
<td>F; 8.7;</td>
<td>F; 8.11;</td>
<td>F; P 5.4.4</td>
<td></td>
</tr>
<tr>
<td>Recurrence</td>
<td>13</td>
<td>F; 8.9.1;</td>
<td>P 8.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar sequence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Futures</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speculative selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pack</td>
<td>I; 14</td>
<td>I; 14</td>
<td>I;</td>
<td>F;</td>
<td></td>
</tr>
<tr>
<td>Expand</td>
<td>I;</td>
<td>I;</td>
<td>I;</td>
<td>I;</td>
<td></td>
</tr>
<tr>
<td>Pipeline</td>
<td>F; 12</td>
<td>I; 12</td>
<td>I;</td>
<td>I;</td>
<td></td>
</tr>
<tr>
<td>Geometric decomposition</td>
<td>I; 15</td>
<td>I; 15</td>
<td>I;</td>
<td>I;</td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>I;</td>
<td>I;</td>
<td>I;</td>
<td>I;</td>
<td></td>
</tr>
<tr>
<td>Category reduction</td>
<td>I;</td>
<td>I;</td>
<td>I;</td>
<td>I;</td>
<td></td>
</tr>
<tr>
<td>Gather</td>
<td>I;</td>
<td>F;</td>
<td>F;</td>
<td>F;</td>
<td></td>
</tr>
<tr>
<td>Atomic scatter</td>
<td>F;</td>
<td>I;</td>
<td></td>
<td>F;</td>
<td></td>
</tr>
<tr>
<td>Permutation scatter</td>
<td>F;</td>
<td>F;</td>
<td>F;</td>
<td>F;</td>
<td></td>
</tr>
<tr>
<td>Merge scatter</td>
<td>I;</td>
<td>I;</td>
<td>F;</td>
<td>F;</td>
<td></td>
</tr>
<tr>
<td>Priority scatter</td>
<td>I;</td>
<td>I;</td>
<td>F;</td>
<td>F;</td>
<td></td>
</tr>
</tbody>
</table>
### Programming Model Support for Patterns

**Table 3.3** Additional patterns discussed. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily).

<table>
<thead>
<tr>
<th>Parallel Pattern</th>
<th>TBB</th>
<th>Cilk Plus</th>
<th>OpenMP</th>
<th>ArBB</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar sequence</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>F</td>
</tr>
<tr>
<td>Futures</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Speculative selection</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Workpile</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Expand</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Search</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Category reduction</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>Atomic scatter</td>
<td>F</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Permutation scatter</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>Merge scatter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
<tr>
<td>Priority scatter</td>
<td>I</td>
<td>I</td>
<td></td>
<td></td>
<td>I</td>
</tr>
</tbody>
</table>